Università
della
Svizzera
italiana

Faculty
of Informatics

Bachelor Thesis

June 24, 2021

Soft bodies simulation

Jacob Salvi

Abstract

Computer graphics is quickly improving to a point in which extremely realistic scenes can be rendered. That said, for simulations it is not sufficient that a scene looks realistic, it is also necessary for them to behave realistically. For this reason it is necessary to create physics based simulations. In reality most objects are not completely solid but can deform to a degree. It is then necessary to model this deformation and simulate it properly. Given that computers have finite resources and operate on finite time steps, we need to create a model that can capture the behaviour of physical systems as accurately as possible, while at the same time being as efficient as possible. To this end, in my project, I have implemented a mass spring system, and tried different integrators, to create a simple soft body simulation that accounts for collision, gravity, and wind.
The project is written in C++ and I have used OpenGL for the rendering.

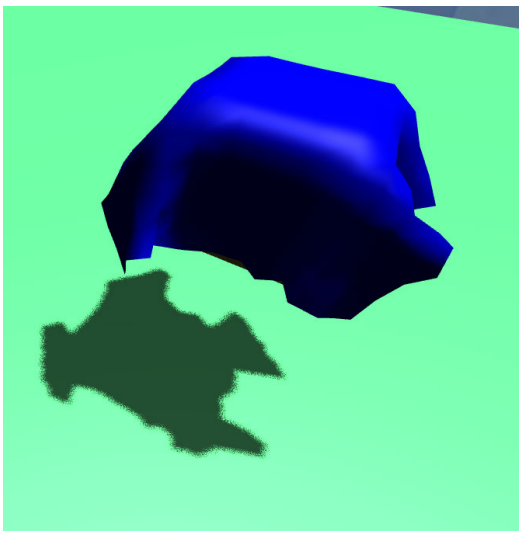


Figure 1. Cloth dropped on a cube

Advisor
Prof. Piotr Didyk

Advisor's approval (Prof. Piotr Didyk): Date: 25.06.2021

Table of contents

1 Introduction

Physics simulation is a developing part of computer graphics concerned with creating a model of a physical system that can closely represent reality. Its applications are wide, going from crude simulation used in computer games and more sophisticated one used for movies to complex realistic simulation needed for surgery simulators, flying simulators and similar. In real life most objects are not completely solid, they can stretch or be compressed to a degree. Therefore to make a realistic simulation we need to be able to handle objects that can deform. In real life if an object moves toward another, eventually the two will collide and there will be a reaction. For example if a ball is thrown against a wall, the wall will not move, but the ball will bounce back. During a collision the objects might deform. An elastic ball that is thrown against a wall will be compressed before bouncing back and restoring its original shape. To make a realistic simulation it is necessary to properly model and handle collisions. And finally, in real life there is friction, an object sliding on a surface does not do so indefinitely, but it will slow down until eventually reaching a full stop. In our simulation, we need to consider friction as well. In this project, I have used the mass spring system to simulate simple deformable objects, such as cloth like structure, deformable spheres and deformable cubes. The mass spring system allows to model an object as a set of particles held together by springs. The simulation includes gravity, wind, friction and some solid primitives, a cube, sphere and plane that can be collided with. The user can interact with the simulation by changing the parameters of the simulation, wind direction and strength, mass of the particles, stiffness of the springs, and by using the mouse to grab and drag the objects.

2 Soft bodies simulation

2.1 Hooke's law

One of the simplest deformable system is a single spring which can elongate and compress. Let us consider how to model a spring. Springs have a resting length l_0. If one end of the spring is attached to an immovable object (for example a wall), and the other one to a particle 2, then we can move the particle horizontally to elongate or compress the spring.

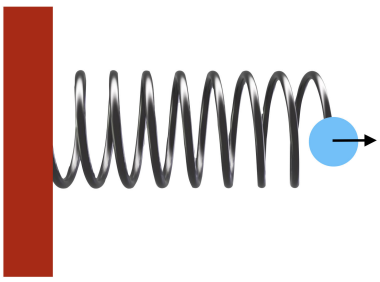


Figure 2. Spring attached to a wall on one end and to a particle on the other

Then, the spring reach a new length x, and it will exert a force to return to its resting state. Hooke's law states that a spring has a force:

$$F_s = k_s \cdot x$$

Where k_s is the stiffness coefficient. In a real system a part of the energy would get dissipated as heat. Therefore we also must consider the damping coefficient $k_d \in [0, 1]$, which models this behaviour. This results in a second force [5]:

$$F_d = k_d \cdot v \cdot x$$

Where v is the velocity of the particle and x, once again, is the displacement.
The total force exerted by the spring, when the particle is moved, is $F = F_s + F_d$.
Let us now consider a more complicated case, and consider a spring that has a particle attached to each side, moving freely (so they do not move exclusively horizontally).

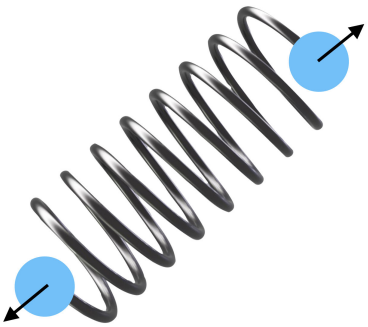


Figure 3. Spring attached to two particles

Let's call the two particles p_0 and p_1 respectively. The two particles have mass m_0 and m_1, and position $\overline{x}_0$, $\overline{x}_1$ where $\overline{x}_0, \overline{x}_1$ are vectors that describes the position in 3D space of the particles, and velocities $\overline{v}_0, \overline{v}_1$.
The spring still has resting length l_0, stiffness k_s and damping coefficient k_d.
Then we get the following forces:

$$F_s = k_s \frac{\overline{x}_1 - \overline{x}_0}{|\overline{x}_1 - \overline{x}_0|} \cdot (|\overline{x}_1 - \overline{x}_0| - l_0)$$

$$F_d = k_d \cdot (v_1 - v_0) \cdot \frac{\overline{x}_1 - \overline{x}_0}{|\overline{x}_1 - \overline{x}_0|}$$

Then the total force acting on p_0 is $F = F_s + F_d$ and the total force acting on p_1 is $-F$ (since it is the same force but in opposite direction).

2.2 Mass springs system

So far we have considered the behaviour of a single spring. Multiple springs can be used to form a system that models an object. An elastic object, can be represented locally as a spring with masses attached to it. Then, the entire object is a system of springs and particles. When a force is applied either to the entire system, as in the case of gravity, or to some particles, as in the case of local collision, the springs will apply a force to the particles, computed as in the preceding section, that will fairly realistically simulate the physical reaction that should be observed. That said, we are left with the problem of how to discretise an object into a mass spring system. Obviously different configuration of springs would make the object behave differently. As an example of how to discretise an object, figure 4 shows how a cloth could be modelled as a system of springs and particles:

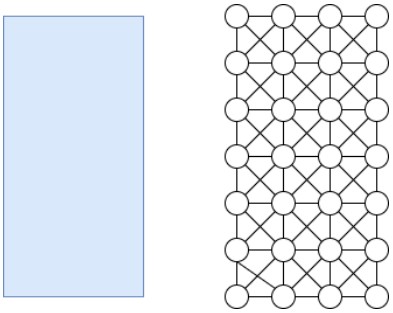


Figure 4. A cloth and its representation as a system of particles and springs

The mass spring system is fairly simple to implement, especially when compared to other techniques such as the finite element mesh, and can give fairly good results. However, it has some limitations. Usually we want to model an object as homogeneous and isotropic. The object should deform the same way in every direction. Given that each spring applies a unidirectional force and has finite size, this is not easy to achieve with a mass spring system. The way it is constructed influences how the system behaves. We also have to fine tune all of the properties of the springs and particles to get a satisfying simulation. The choice of coefficients is not always obvious. Often a trial and error approach is needed to set them. Another weakness of the mass spring system is the fact that it does not capture volumetric effects, such as conservation of volume. This problem can be mitigated to a certain extent by having particles inside the object as well as on its surface (for example creating a tetrahedral representation of the object) [5] [6].

2.3 Integrators

When simulating the system we want to know where each particle will be at a time t. Given a particle that is at position x_0 at the beginning of the simulation, its change in position is given by its velocity v:

$$x(t) = x_0 + \int_0^t v(t)dt$$

The velocity of the particle may change over time due to an acceleration. Then, we must find the velocity at time t. Given initial velocity v_0 and acceleration a, we can calculate the change in velocity as:

$$v(t) = v_0 + \int_0^t a(t) \cdot dt$$

Since the particles are subject to forces, we can find the acceleration by applying Newton's law:

$$a(t) = \frac{F(t)}{m},$$

where $F(t)$ is the force acting on the particle at time t, and m is the mass of the particle.
To simulate the system in real time we need a way to compute the time integration as accurately as possible and as efficiently as possible, given that computers have finite time steps.

2.3.1 Explicit Euler integrator

The simplest integrator that solves the above system is the explicit Euler integrator in which we approximate the derivatives as follows:

$$\dot{v} = \frac{v(t+\Delta t) - v(t)}{\Delta t} + O(\Delta t^2)$$

$$\dot{x} = \frac{x(t+\Delta t) - x(t)}{\Delta t} + (\Delta t^2)$$

Given that $\dot{v} = a$ and $\dot{x}(t) = v$ we get:

$$v(t+\Delta t) = v(t) + \Delta t \cdot a$$

$$x(t+\Delta t) = x(t) + \Delta t \cdot v$$

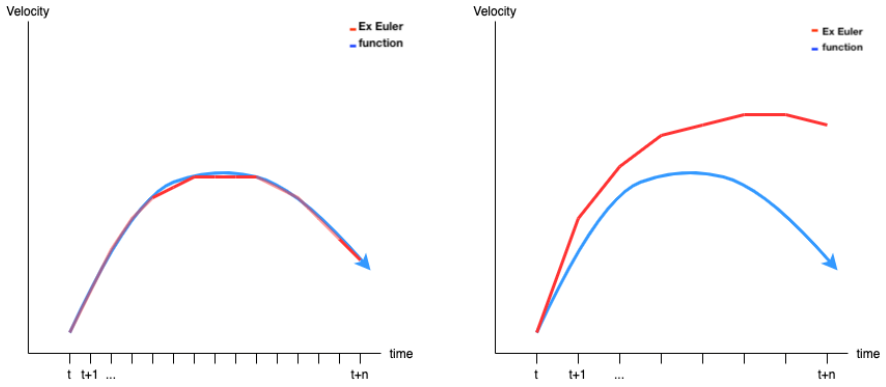
Since we already have $v(t+\Delta t)$ when we compute the position we can modify the function to get $x(t+\Delta t) = x(t) + \Delta t \cdot v(t+\Delta t)$. Then for the explicit Euler integrator, at each step, we calculate the force applied to every particle (spring forces, gravity, wind, friction, etc...), and use it to compute the acceleration that is needed to update its velocity. Then we update the particles position. Since we compute the quantities of the next step directly from what we know about the particle at the current time using an explicit formula this scheme is called explicit Euler. This integrator is easy to implement but it is not very stable (figure 5), since we assume that the velocity and acceleration remain constant during a time step. If Δt is large then the integrator destabilizes, therefore explicit Euler needs small times steps, of the order of milliseconds or tenth's of milliseconds. We want to run the integrator as many times a second as possible. However, if an iteration of the integrator is slower than the time step it is using, it will not work well for a real time simulation. Then we must choose the smallest time step that is large enough for one iteration of the integrator. In addition to this, the mesh of the object must be updated in time for the rendering of the scene, that said it would be costly to update the mesh every time we update the particles. We need to update it in time for a new frame to be rendered. Then given target frame rate fr and time step Δt, for the integrator, we compute :

$$n = \left\lfloor \frac{1}{fr \cdot \Delta t} \cdot 1000 \right\rfloor$$

Every n iteration of the integrator we can update the mesh of the object, just in time for the new frame to be rendered. In pseudocode explicit Euler looks like this:

Algorithm 1 Explicit Euler

```
while simulating do
    for p in particles do
        p →setForce(envForce + collForce+frictionForce + cumulative springs force)
        p →set velocity( p → Velocity+ΔT · p →Force/p →Mass)
        p → set position(p → position+ΔT · p →velocity);
    end for
    if i%n == 0 then
        i=0
        update the scene
    else
        i+=1
    end if
end while
```



(a) With small time steps the integrator is stable

(b) With bigger time steps it stops being accurate

Figure 5. Visualization of the stability of the explicit Euler integrator.

In the above pseudocode envForce is the force due to the environment (gravity, wind), collForce is the force due to collision, frictionForce is the force due to friction and cumulativeSpringsForce is the sum of the forces of all the springs that act on that specific particle.

2.3.2 Runge Kutta

Explicit Euler is first order order accurate, halving the time step halves the approximation error. It is possible to do better using Runge Kutta. The second order Runge Kutta integrator is second order accurate, meaning that when the time step is halved the error is reduced to one fourth. This integrator works by evaluating the forces two times per each time step.
Given time step Δt we first find the velocity at time $\frac{\Delta t}{2}$ and use it to find the force at time $\frac{\Delta t}{2}$. We then use these updated values to update the position and velocity of the particle. The algorithm is as follow:

$$c = \frac{f(x(t), v(t))}{m}$$

$$b_1 = v(t) + \frac{\Delta t}{2} \cdot c$$

$$b_2 = f(x(t) + \frac{\Delta t}{2} \cdot v(t), v(t) + \frac{\Delta t}{2} \cdot c)/m$$

$$x(t + \Delta t) = x(t) + \Delta t \cdot b_1$$

$$v(+\Delta t) = v(t) + \Delta t \cdot b_2$$

Where f(x, v) is the force exerted on the particle given its position x and velocity v.
c, b_1, b_2 are needed to keep track of the variables. Their name have no particular meaning.
Since we are evaluating the forces twice, this integrator takes as much time as running explicit Euler two times. But

since it is second order accurate it is better since we can either use a bigger time step and get the same results as with Euler, or use the same time step and get better results.
In pseudocode it is as follow:

Algorithm 2 Second grade Runge Kutta

while simulating **do**
    a1 = []
    a2 = []
    b1 = []
    b2 = []
    for i = 0 to number of particles **do**
        a1.append($p[i] \rightarrow$ velocity)
        a2.append((envForce + collForce+friction + cumulative springs force)/$p[i] \rightarrow$ mass)
        b1.append($p \rightarrow$ velocity+$\frac{\Delta T}{2} \cdot a2[i]$)
        b2.append((envForce + collForce+friction + cumulative springs force)/$p[i] \rightarrow$ mass)
        $p[i] \rightarrow$set velocity($p[i] \rightarrow$ Velocity+$\Delta T \cdot b2[i]$)
        $p[i] \rightarrow$ set position($p[i] \rightarrow$ position+$\Delta T \cdot b1[i]$);
    end for
    if $i\%n == 0$ **then**
        i=0
        update the scene
    else
        i+=1
    end if
end while

The last integrator I implemented is the fourth grade Runge Kutta integrator, that as the name implies, is fourth grade accurate, meaning that halving the time step will reduce the error by a factor of 16.
The fourth grade Runge Kutta integrator works by evaluating the forces 4 times, and therefore taking four times the time of explicit Euler to execute.
In figure 6 we demonstrate the difference between the three integrators for the function e^x:

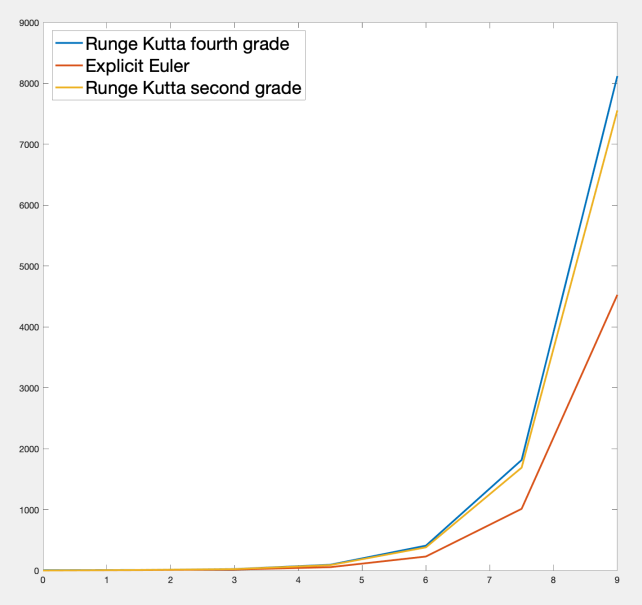


Figure 6. Comparison between, explicit Euler, Runge Kutta second grade and Runge Kutta fourth grade

3 Collision detection and response

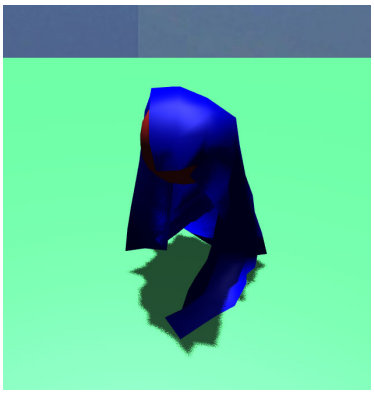
Two objects can not occupy the same space at the same time. To ensure that this hold true it is needed to run a collision detection at every frame, and if a collision occurs, it has to be handled. In my simulation, I address collisions between some shapes and the mass spring system. The simplest collision handling can be done for a sphere, due to its very simple geometric definition. Therefore, I will first describe how to handle the collision with a sphere, and then, extend it to more complicated shapes.

3.1 Collision with a sphere

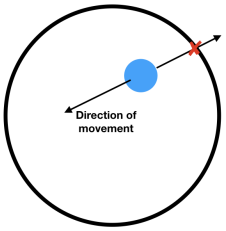
There are two issues that have to be addressed when solving for the collision. First, we need to decide whether two objects collide. Second, if they do collide we need to compute new forces acting on them. The deformable object is made of particles held together by springs. If any of the particles is inside the sphere then a collision is happening. To resolve it, we must apply a force to the particle that will push it outside the sphere. The force should depend on the mass of the objects, direction of motion, and velocity. I simplified the problem using a rough approximation of the force. I assumed that the force, applied to the object, due to collision is proportional to the penetration of the particle in the object. To compute it, we create an intersection ray that goes in the opposite direction of the movement of the particle and with the particle position laying on it (figure 7). We compute the intersection between this ray and the surface of the sphere, and call this point the intersectionPoint. Then the force that we apply to the particle is:

$$f = \mu \cdot (intersectionPoint - particlePosition)$$

Where μ is a constant. This force is proportional to the penetration of the particle in the sphere and will push it in opposite direction of its movement, placing the particle at the point it penetrated the sphere.



(a) collision between a cloth and a sphere



(b) collision response, the red X represents the intersection with the sphere surface

Figure 7. Visualization of collision detection and response

3.2 Collision with a plane

A plane is a two-dimensional object; therefore we can not check whether a particle is inside it. Instead, one can verify the proximity of the particle to the plane. For collision detection, we add a small depth to the plane, making it effectively a box, so that we can check whether a particle is inside of it. The rest of the collision works the same as for a sphere. We calculate the intersection point using a ray in opposite direction of the particle movement and use it to compute a force to apply to the particle to push it away from the plane.

3.3 Collision with a cube

A more complicated situation happens when the object that is being collided has some corners or edges, in such cases it is possible that no particle is inside the object but a collision between a corner/edge and a triangle formed by three particles is happening. This could happen for example while colliding a cube (figure 8).
To solve this problem, in addition to checking whether any particle is inside the cube, we have to check whether any one of the triangles that form the faces of the cube is intersecting with any of the triangle that forms the mesh of the

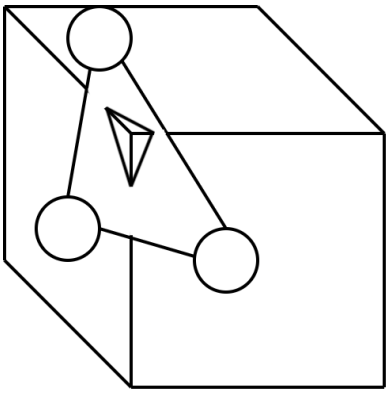


Figure 8. Collision with a corner of the cube

deformable object 9.
To check whether any vertex of the cube-triangle is penetrating the triangle formed by three particles, we need to check whether any two of its edges intersect with the triangle formed by the three particles. If such two intersection exist we can compute the middle of those two intersection points and set it as the true intersection point.
The force applied to the particles has to be proportional to the distance between them and the intersection point. Since the particles are the vertices of the triangle, we can use the barycentric coordinates. Therefore the force that we must apply to each particle is:

$$f = \mu \cdot \lambda \cdot (corner - intersectionPoint)$$

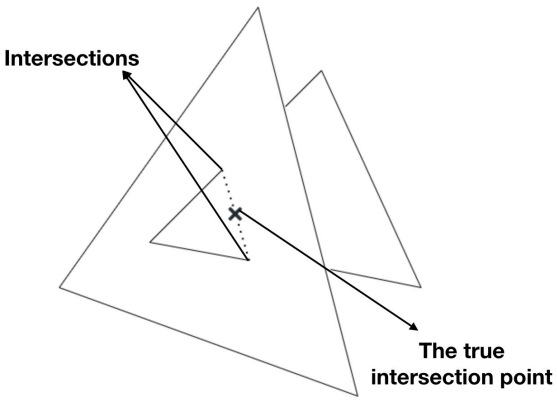


Figure 9. collision between two triangles, the cross is the middle point between the collision with the sides of the second triangle

As before μ is a constant and λ is the barycentric coordinate of the intersection point with respect to the particles

In this case, the response force is proportional to depth of the collision and to the barycentric coordinate of the intersection point. Since the intersection point is the middle point between the two edges of the triangle I am assuming that locally in time the velocity of the moving object is constant.

If instead, the cube triangle and the particle triangle intersect side way (figure 10) then we have to check if any of the three cube-triangle sides is penetrating the other triangle.
In this case, we compute the orthogonal projection between the intersectionPoint and the side of the cube triangle in opposite direction of movement. And as before we can use these two points to compute a force to apply to the particles.

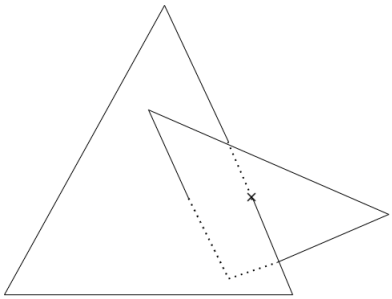


Figure 10. Two triangle colliding sideway

We can observe the results of collision detection in figure 11. In which, a cloth is dropped on a solid cube. The only forces acting on the cloth are gravity and collision.

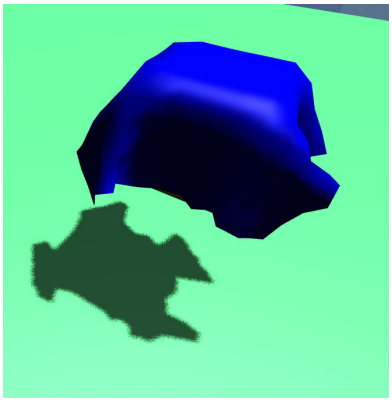


Figure 11. cloth-cube collision

3.4 Friction

Without friction the objects would simply keep sliding in whichever direction they are going even when colliding, therefore to get a more realistic simulation it was needed to implement it.
Friction due to gravity, is a force in the opposite direction of movement proportional to the cosine of the angle between gravity and the inclination of the slope the object is sliding on 12. It can be calculated as follow:

$$F_{fr} = \mu \cdot m \cdot g \cdot \cos(\theta)$$

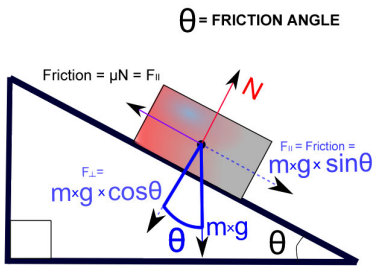


Figure 12. friction [1]

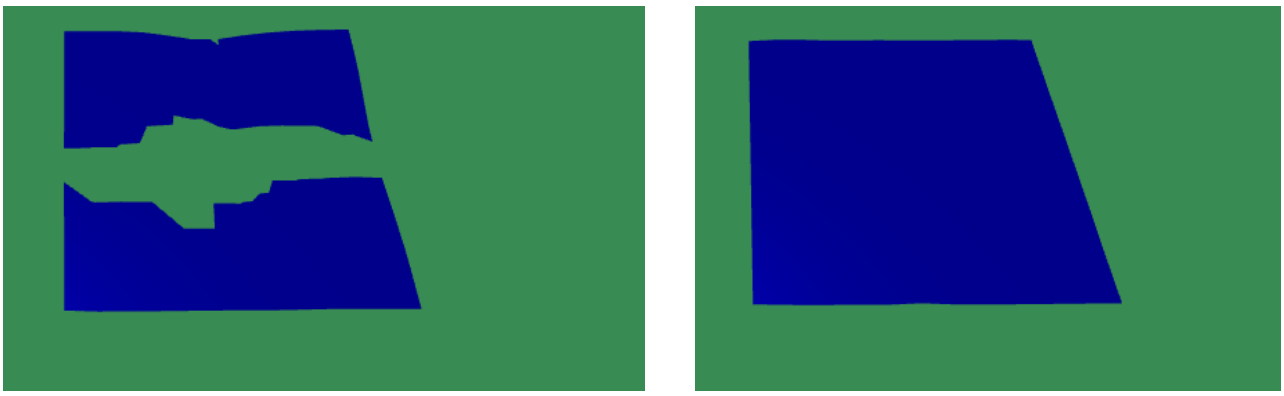
During the collision detection, if a particle collide with an object, a part from a collision force, a friction force is also added. In case of triangle-triangle collision the friction is distributed to the three particles forming the triangle according to the barycentric coordinates of the collision point on the triangle.
In pseudocode collision detection can be implemented as follows:

Algorithm 3 Collision detection

```
for i in deformable objects particles do
    if i inside object then
        i → set force (μ · (intersectionPoint − i → Position))
        i → add force (frictionConstant · gravity · cos(angle))
    end if
end for
for tri in deformable object mesh do
    if tri and object intesect then
        for
            doi → add force(barcoord · constant · (intersectionPoint − corner))
            i → add force(barcoord · frictionConstant · gravity · cos(angle))
        end for
    end if
end for
```

3.5 Visual problem

Regardless of which shape is intersected, if we push the particle on the object's surface, then there would be some visual problems. If three particles that form a triangle of the mesh of the cloth are exactly on the surface of the colliding object, then the triangles of the two meshes overlap. In such a situation OpenGL can not properly decide which one of the two objects is closer to the camera, and therefore which one should be shown. The result is that some portions of both are shown intermittently (figure 13).
To solve this problem, I added a small padding to the object to push the particle slightly outside it. The following figure showcase the problem:



(a) pushing the particle on the surface of the plane

(b) pushing the particle to the surface and adding a small padding

Figure 13. Collision with and without padding

4 Deformables tridimensional objects

I have implemented three deformable objects, cloths, deformable sphere and deformable cubes.
The cloth is made of a simple grid of particles and springs. Any three particle, in counterclock wise order form one of the triangle needed for the rendering in OpenGL. The particles are connected to every neighbouring particle, see figure 4.

To represent the surface of the sphere as formed by triangles we need to divide the sphere in sectors (longitude) and stacks (latitude), figure 14. Every combination of stack and sector corresponds to a point on the sphere surface. We can put particles at these positions thus obtaining an approximation of a sphere. A variable "size" controls how many stacks/sectors are there, allowing to create a sphere defined by more or less points. The more points it has the better it approximates a sphere, also the more expensive the update become, since more particles have to be updated. Similarly as for the cloth, three particles in counter clock wise order form a triangle of the mesh. Every particle on the sphere is connected to every other particle by a spring whose resting length is the distance between

the two particles.

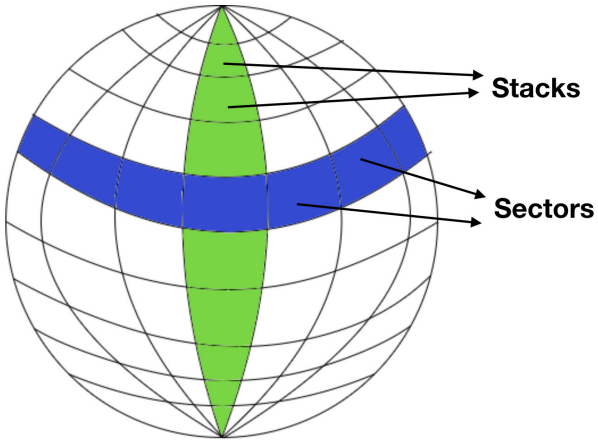
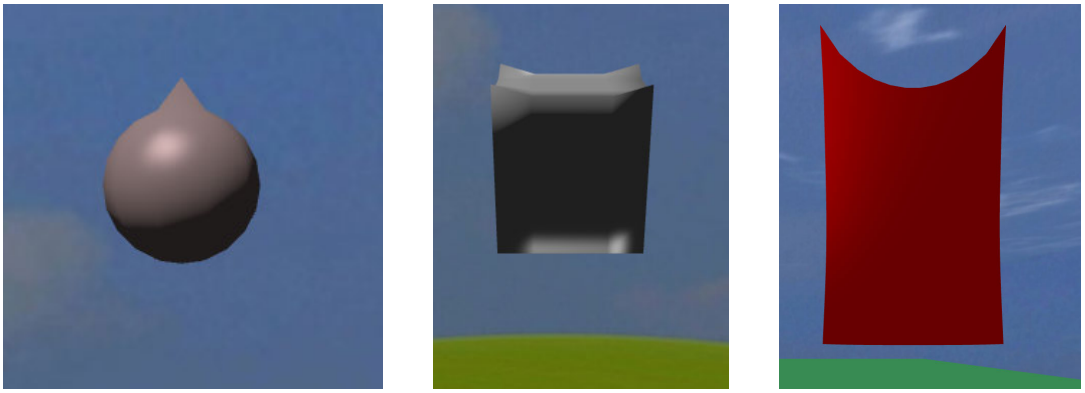


Figure 14. Representation of the sphere for OpenGl

The cube is formed by sub-cubes and a variable "size" allows to decide how many sub cubes to use. At each vertex of each of the sub-cubes we create a particle, meaning that the cube has particles inside as well, but only the ones on the surface are used to form the mesh for OpenGL. Similar to the sphere, every particle is connected to every other. Naturally, the more sub-cubes are used the more convincing the simulation looks, but also the more expensive it becomes.

To demonstrate the simulation, for the purpose of this report, I consider all objects hanging. During the simulation they can be dropped by pressing the "drop" button in the GUI (figure 15).



(a) Deformable sphere in the resting position

(b) Deformable cube in the resting position

(c) Cloth in the resting position

Figure 15. Visualization of the deformable object while they are hanging

4.1 3D models

Simple geometrical objects can be a little dull. To make the scene better I have adapted a script, that professor Didyk Piotr provided to import 3D models. I converted it from JS to C++.
The script parses .obj files and creates arrays for vertices, UV coordinates, normals and indices. At each vertex I put a particle, making sure there are not duplicates, and then connect them with springs. Using this script, for example, we can import a teapot 16

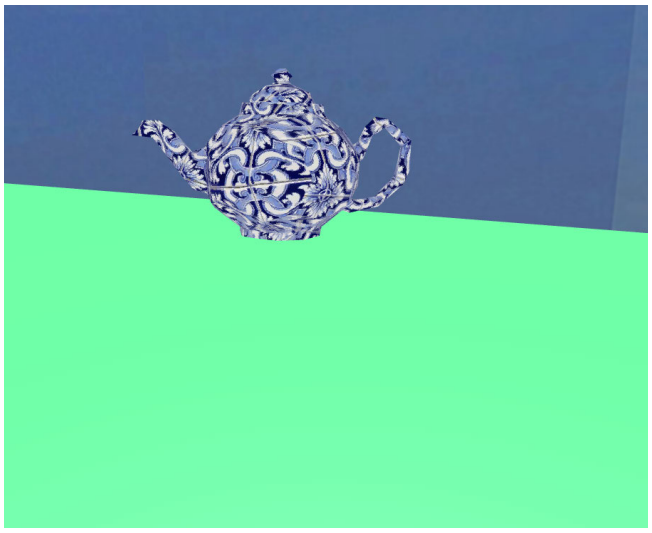
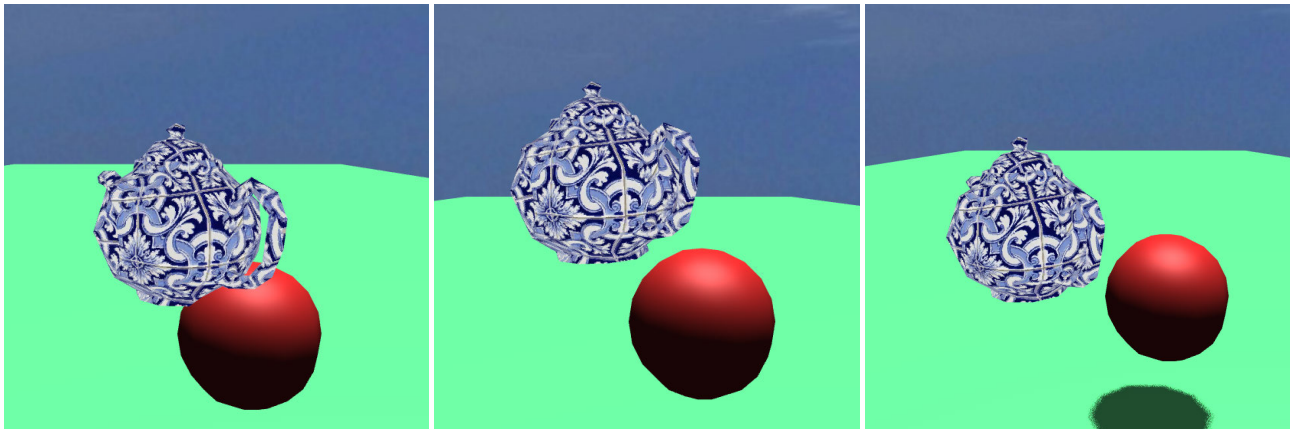


Figure 16. Teapot in its resting position

Models can be quite detailed, often formed by a fairly high number of polygons. This can be a problem since an excessive number of particles and springs slows down the simulation to an unacceptable extent for a real time application. Therefore, it was necessary to simplify the models before using them. Blender, a popular 3D modelling application, allows to reduce the polygon count, and therefore the vertex count, of a model before exporting it.
Once the model has been imported it behaves has expected. It deforms correctly and it interacts with the other objects as it should. Below we can observe a collision between the teapot and a solid sphere (figure 17)



(a) Impact between the teapot and sphere (b) The teapot reacts by bouncing back (c) The teapot then keeps falling down

Figure 17. Collision between a teapot and a solid sphere.

5 Bounding volume hierarchy

Given that the deformable objects can have arbitrary, non necessarily convex, shapes during the simulation it can become quite expensive to check for collision agains them. Checking for collision between each pair of objects at each frame would greatly lower the frame rate of our simulation. To solve the problem we can put the deformable objects inside some simple geometric shape that can quickly be checked for collision. Then we need to run the collision detection with the actual object only if the enveloping shape detects a collision[3]. We can take this a step further by creating a tree in which the root contains the whole deformable object and the children contains subsection of it, recursively until a leaf criterion is met. This criterion is usually that either the leaf contains a single triangle of the mesh object or that it contains a small number of the particles that forms it. This approach is called Bounding volume hierarchy (that from now will be referred to using the acronym BVH)[4].
With a BVH we can speed up collision detection. Starting from the root if we do not collide with the shape that encompasses the object we can terminate the collision detection. Otherwise, we check recursively for collision against its children until a leaf is met, in which case we have to check for collision against the subsection of the object that

it contains. Pseudocode:

Algorithm 4 Collision(BVH, object)

```
1:  if not collisionDetection(BVH → Sphere, Colliding Object) then
2:      return False
3:  else
4:      if obj → BVH is a leaf then
5:          check for collision with the subsection of the object present in the leaf
6:          ...
7:      else
8:          for i in BVH → children do
9:              Collision(i, object)
10:         end for
11:     end if
12: end if
```

There are many different BVHs such as axis aligned boxes, spherical BVH, convex hull, etc, I have chosen to use spherical ones for their simplicity (figure 18).

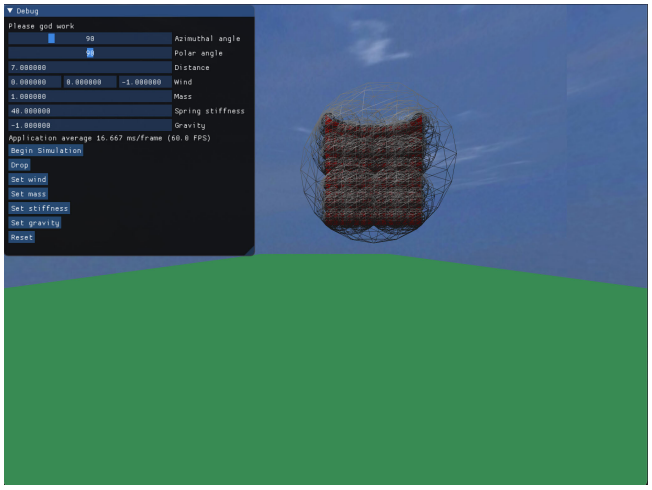


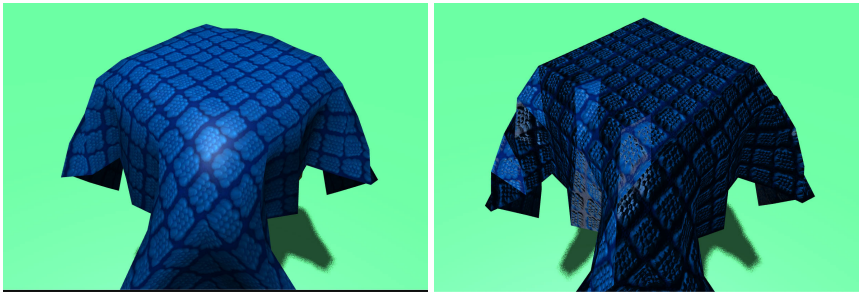
Figure 18. Spherical BVH

It is a normal practice for BVH to be either quad-trees or oct-trees. In my specific case I implemented them as quad-trees.

6 Rendering in OpenGL

To render the scene, I have used OpenGL. I have based some of my code on the following tutorials [2]. OpenGL needs the coordinates of the vertices of a triangle to be able to draw it. Then when creating the objects I have mapped every three particle on their surface to the vertex of a triangle. When the particle position is update the corresponding triangle vertex is also updated. Given that the object are moving and that they might collide with other objects I implemented shadows mapping that, other than making the scene looks better, gives a sense of distance between the objects. Finally to make the scene look better I have implemented a skybox.

The cloths can have textures applied to them. Normally textures look flat, to create the illusion that there are some bumps and cavities I have implemented normal mapping (figure 19).



(a) Texture without normal mapping (b) Texture with normal mapping

Figure 19. Texture with and without normal mapping

7 User interaction

The simulation allows the user to interact with the system. Beside moving the camera to look at the scene from a different perspective, the user has access to a simple GUI and to the possibility to grab the various objects in the scene with the mouse.

7.1 GUI

To implement the GUI, figure 20, a library called IMGUI has been used. The GUI allow the user to set gravity, wind strength and direction, mass of the particles, coefficients of the springs. From the GUI the user can also reset the simulation and drop the objects.

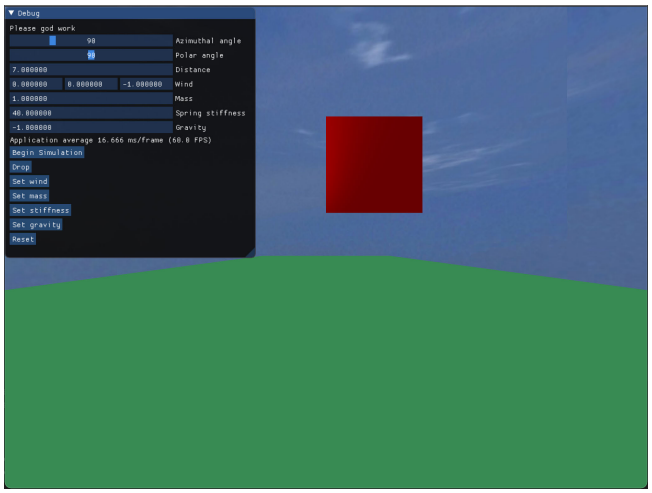
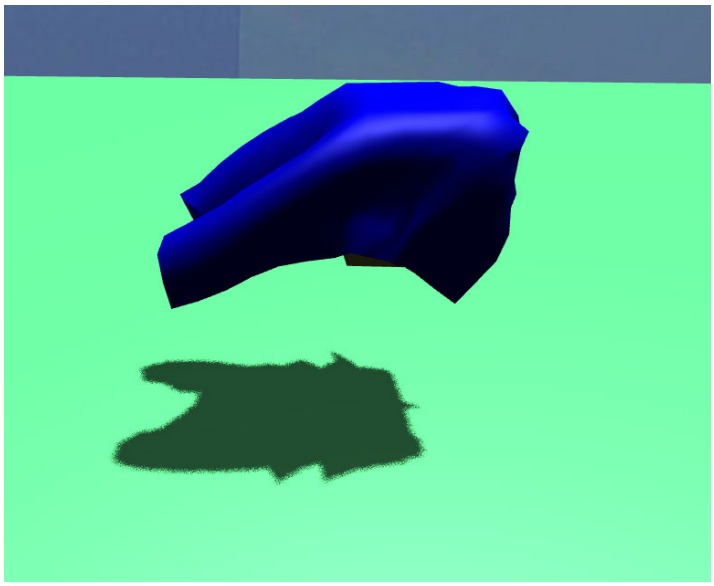
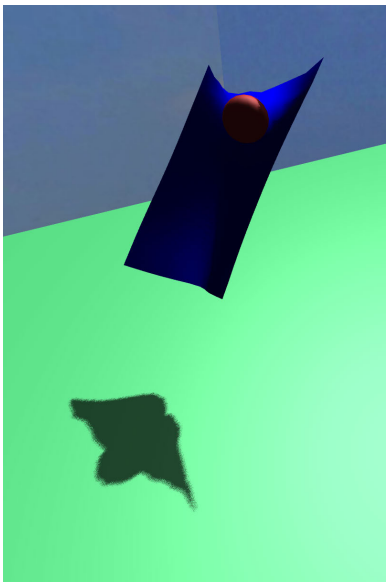


Figure 20. GUI

To set the wind it is sufficient to change the x, y and z values in the wind section and then click the "set wind" button. The wind acts as a force, in the direction chosen, applied to every particle. The figure 21 shows its effect. On the left we can see a cloth dropped on a cube that is also being blown by the wind, on the right an hung cloth is being blown against a sphere:



(a) Cloth on cube being blown by wind

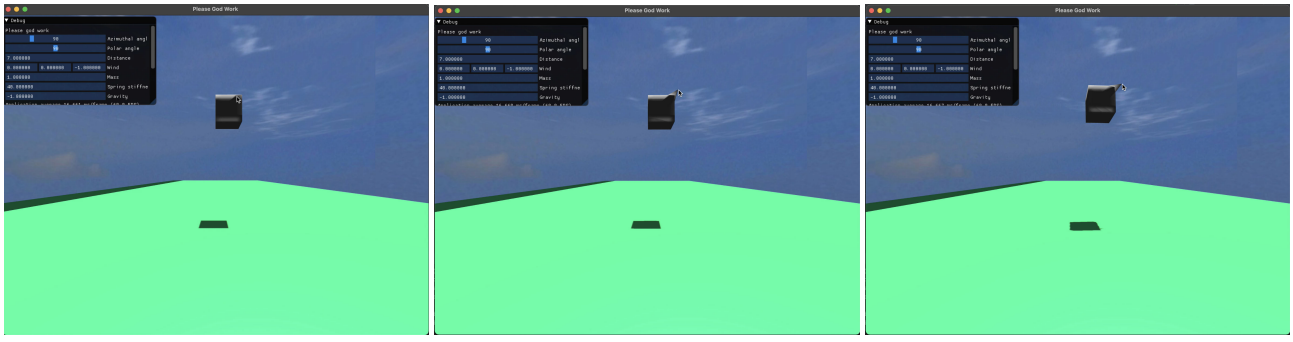


(b) Cloth being blown against a sphere

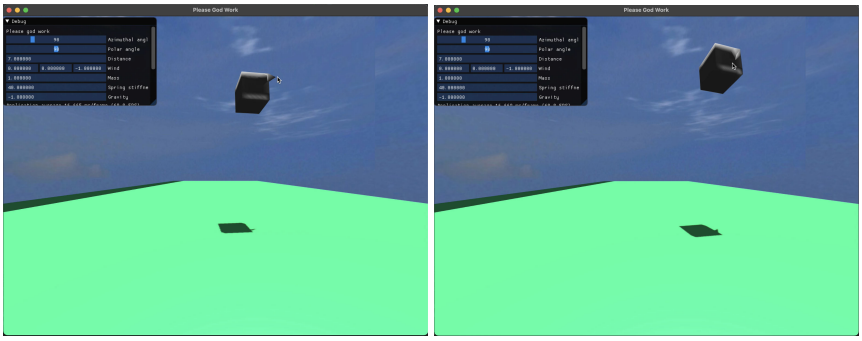
Figure 21. Wind

7.2 Mouse picking

Clicking on an object should allow the user to grab it such that a subsequent movement of the mouse allows the user to drag the object around (figure 22).



(a) (b) (c)



(d) (e)

Figure 22. (a) Clicking on a deformable cube (b) Begin dragging the object (c) The cube start reacting to the drag (d) Reaction e) Release of the left mouse button, the cube still has some momentum

To achieve this effect I have used ray-casting. The window created by OpenGL has a width and height expressed in pixels. The x coordinate increases going right and the y coordinate increases going down. Then, we can address each pixel as a pair of x, y coordinates.
When the mouse is clicked, we can get the position of the pixel it is hovering as (x, y) 23.

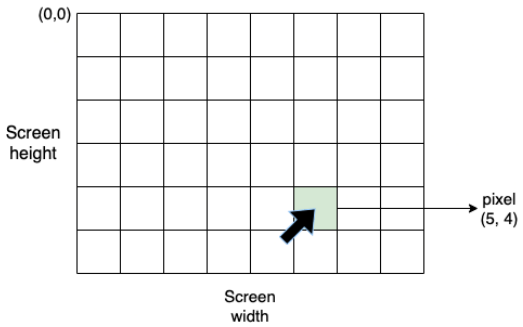


Figure 23. Mouse clicking on the screen

Now we need to transform it from screen coordinates to 3D normalized device coordinates.
To that end, we use the following equations:

$$x = \frac{2 \cdot x}{screenWidth} - 1$$

$$y = 1 - \frac{2 \cdot y}{screenHeight}$$

Where screenWidth and screenHeight are the width and height of the screen expressed in number of pixels.
At this point we need to transform the vector in camera coordinates. First we create a four dimensional vector v=(x, y, -1, 1). We set the z to -1 since in OpenGl it correspond to the forward z direction.
Multiplying this vector by the inverse of the projection matrix gives us the vector in camera coordinates:

$$v = ProjectionMatrix^{-1} \cdot v$$

Finally, by multiplying v by the inverse of the view matrix we get the ray in world coordinates:

$$v = ViewMatrix^{-1} \cdot v$$

Let us call this ray the "mouse ray". After normalizing it, we can check whether it intersects any object.
For each object we multiply the mouse ray by the inverse of the object model Matrix to transform it in object coordinates and do a ray object intersection. If the ray does not intersect, we check the next object, otherwise, if an intersection is found, we need to save the point of intersection as the intersectionPoint.
If the object we are intersecting is a solid then we save the intersectionPoint as the actual point of intersection, if it is a deformable object we need to find the closest particle to that point and save that as the intersectionPoint.
Now that we have got hold of the object we are faced with another problem, when moving the mouse where should the object be moved to?
The object should move on a plane parallel to the screen (figure 24), then the plane normal has to be the view direction. Also this plane must pass trough the intersectionPoint, otherwise the object would jump to the plane when we click on it. Let us call this plane the translation plane. To drag the object, we need to shoot a new mouse ray, at regular intervals of time, and check in which point it intersects the translation plane. Once this point has been found we can update the position of the grabbed object such that the point on the object grabbed by the mouse will lay on the intersection between the mouse ray and the translation plane.
Dragging the object will also start the simulation (the integrator will start running), and free the object (since the deformable objects are initially hung). From the figure 22 it can be observed that the object reacts to the dragging fairly well, by swinging and moving in the direction of the drag.

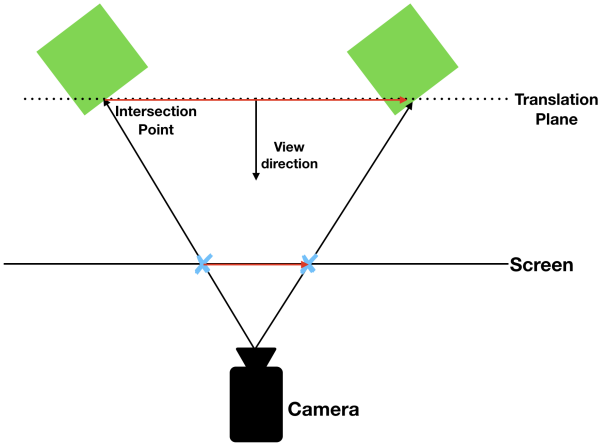


Figure 24. Visualization of mouse picking. The two blue X represent the pixels being hovered by the mouse when picking and dragging.

If the mouse ray intersects multiple objects the closest one to the camera is the one that gets dragged.

7.3 Optimization

Similarly to the collision detection also the ray intersection is an expensive operation for an arbitrary non necessarily convex object. In the same way that we used the BVH to speed up collision detection it can be used here as well to speed up the ray intersection. To this end, we check against the root of the BVH for an intersection and if there is none we terminate the procedure. Otherwise we check recursively until a leaf is met, at which point we can do a simple check against a small subsection of the object.

8 Conclusion

In this project I have implemented a simple mass spring system that simulate soft bodies in a physical environment. I have experimented with different integration schemes taking in consideration both how accurate they are and how they perform. I have studied the techniques necessary to detect collisions and to respond to them, and then, I have implemented such solutions. This project also gave me the chance to strengthen my C++ skills and to learn OpenGL.

8.1 Benchmark

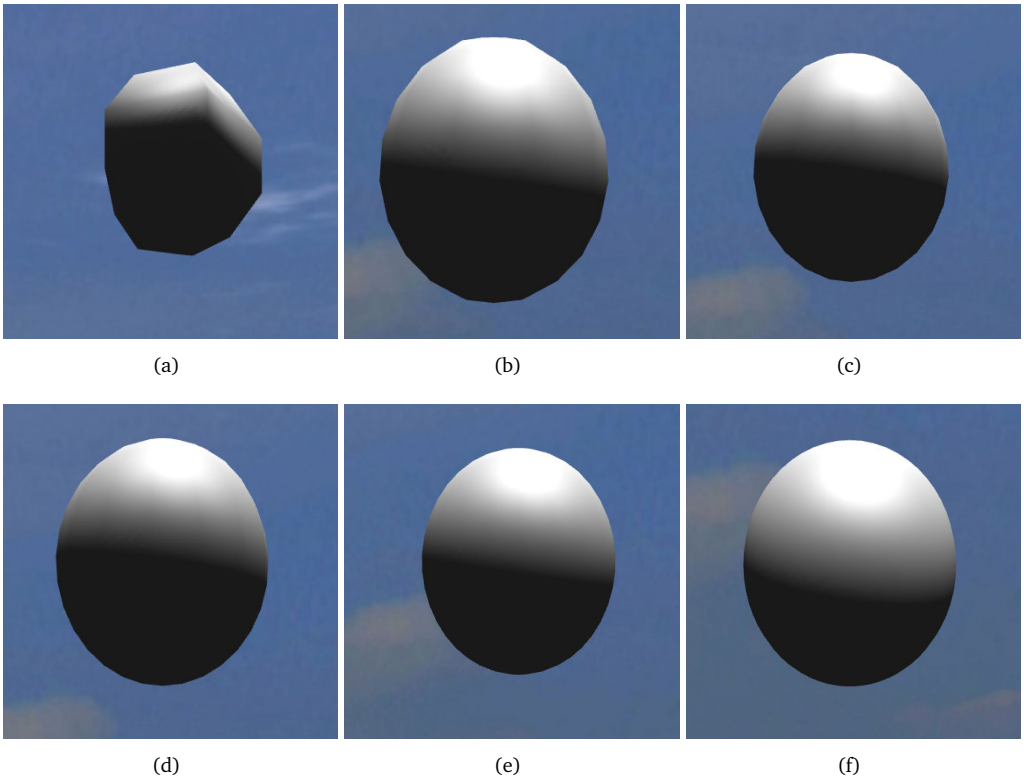
I have used a separated thread dedicated to running the integrator for the deformable objects. This allowed the frame rate to be constant since the main thread is not affected by the simulation. That said, if updating the position of all of the particles that form an object is slower than 16 milliseconds, the simulation can not update in time for the repainting of the canvas. Also in the section dedicated to the integrators we said that we should run the integrator with time steps of around one millisecond, meaning that to function, one iteration of the integrator should be faster than one millisecond. For the benchmark we can investigate how quickly the Runge Kutta fourth grade integrator (the one that gives the least error/most realistic simulation) runs for different sizes of our objects. The benchmark ran on a Macbook Pro 2018, running macOS Big Sur version 11.2.1. At the time the kernel being used was Darwin 20.3.0. The code has been written in C++ 11 and the compilation has been done using the -O2 optimizer.

8.1.1 Sphere

The performance depends on the complexity of the mass-spring system. The size of the sphere influences by how many particles and springs the sphere is formed. For the benchmark I have kept track of the time required for one iteration of the fourth grade Runge Kutta integrator to complete for spheres of different sizes. From the below table we can see that using a size bigger than 12 breaks the 1 average milliseconds per update of the sphere, and a sphere of just size 30 becomes pretty slow to update, requiring almost 40 milliseconds. From the figure 25 we can observe how the size influence the sphericity of the sphere. Going from size 5 to size 10 dramatically improves the shape of the object, increasing the size to 12 or 15 gives an even better shape but the difference between 15 and 20 or 20 and

30 is not as well defined or easy to spot. Then the best balance between size and performance is meet at a size of around 12.

Size	#Particles	# Springs	Average Time (microseconds)
5	22	231	80
10	92	4186	900
12	134	8911	1500
15	212	22366	2600
20	382	72771	8000
30	872	379756	38000



(a) (b) (c)

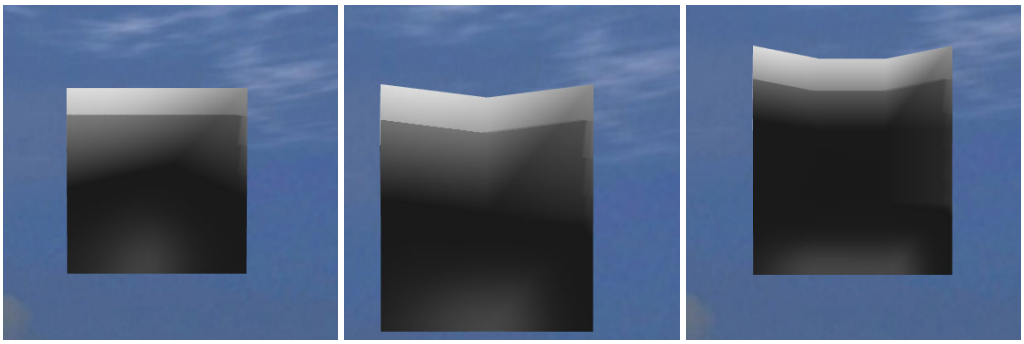
(d) (e) (f)

Figure 25. a) sphere of size 5 b) size 10 c) size 12 d) size 15 e) size 20 f) size 30

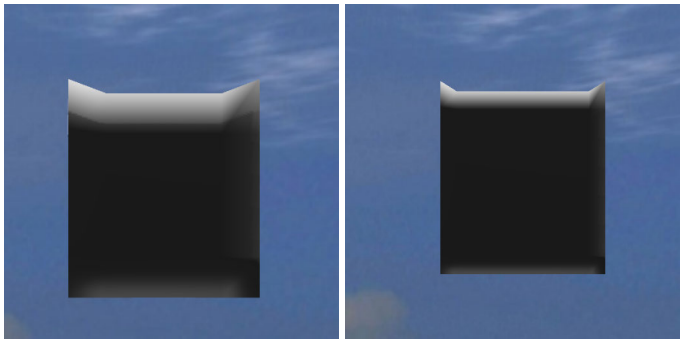
8.1.2 Cube

The size of the cube controls how many sub-cubes it is formed of. The more sub-cubes the more particles and springs are present in the system. Similar to the test for sphere, I have investigated the time required for one iteration of the fourth grade Runge Kutta. To appreciate the change in size we need to take a look at the cube while it is hanging, figure 26. A cube of size 1 does not seem to deform at all, the more the size increases the more natural it looks. That said, from the table, we can see that at a size of 4 we are already breaking the 1 millisecond mark, and at size 10 the time needed for one run of Runge Kutta, 86 milliseconds, is way too much for a real time application. Then settling with a size of 3 or 4 would be best.

Size	#Particles	# Springs	Average Time (microseconds)
1	8	28	30
2	27	351	100
3	64	2016	400
4	125	7750	1200
5	216	23220	2700
6	343	58653	6400
7	512	130816	14000
8	729	265356	25000
9	1000	499500	48000
10	1331	885115	86000



(a) (b) (c)



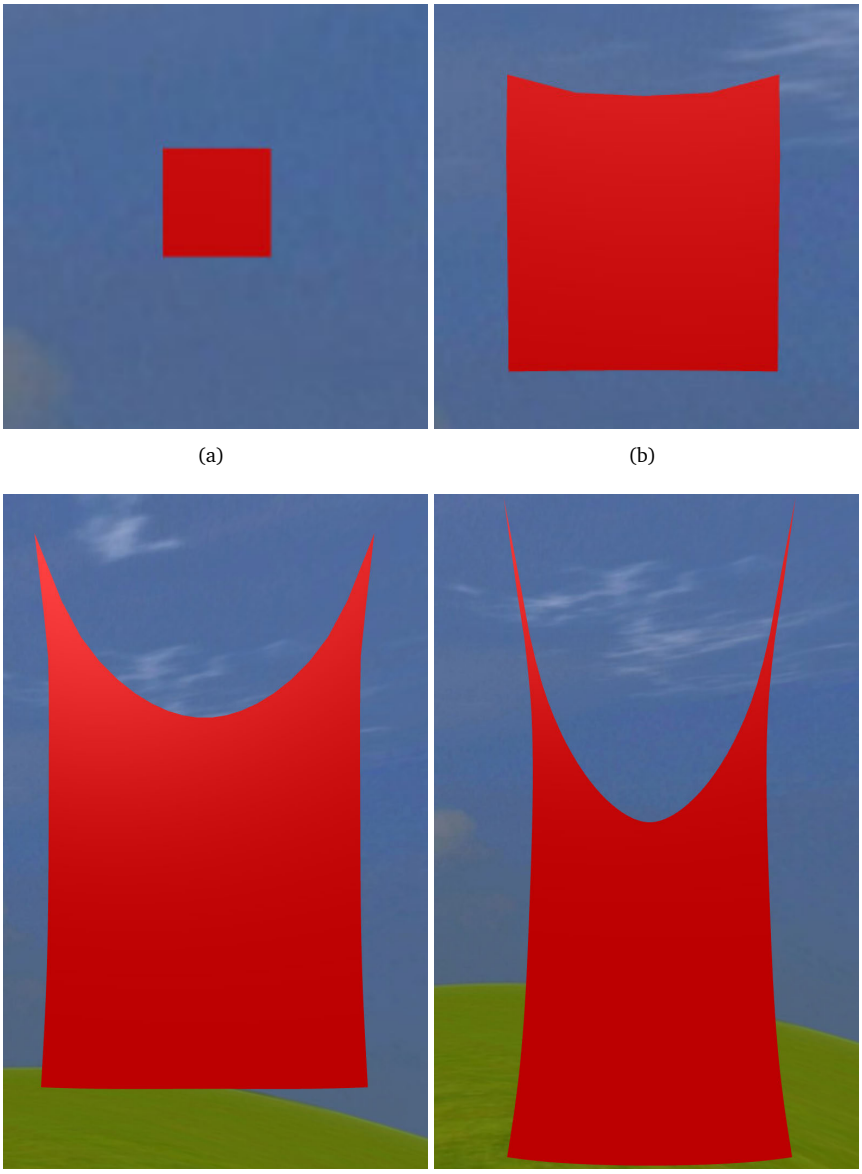
(d) (e)

Figure 26. a) Size 1 b) size 2 c) size 3 d) size 5 e) size 10

8.1.3 Cloth

Cloths can be created in any ratio, but for the sake of simplicity of the benchmark we will consider only square cloths. To better appreciate the difference in particle density we will look at the cloths in their resting position, figure 27. The cloth of size 2 looks like a square, at size 5 we can see a bit of the deformation, and at larger size we can properly observe how the cloth would behave. We can notice that we break the one millisecond per iteration of Runge Kutta between 30 and 40 of size, and even with a size of 50 the time is reasonable.

Size	#Particles	# Springs	Average Time (microseconds)
2	4	6	20
3	9	20	30
5	25	72	50
10	100	342	130
20	400	1482	400
30	900	3422	900
40	1600	6162	1100
50	2500	9702	1400
100	10000	39402	6000
200	40000	158802	41000



(a) (b)

(c) (d)

Figure 27. a) Size 2 b) size 5 c) size 20 d) size 50

8.1.4 Benchmark conclusions

To properly understand the benchmark, first we need to consider the computational complexity of one iteration of the Runge Kutta fourth grade integrator.

Algorithm 5 Fourth grade Runge Kutta

1: a1 = []
2: a2 = []
3: b1 = []
4: b2 = []
5: c1 = []
6: c2 = []
7: d1 = []
8: d2 = []
9: **for** i = 0 to number of particles **do**
10: a1.append($p[i] \rightarrow$ velocity)
11: a2.append((envForce + collForce+friction + cumulative springs force)/$p[i] \rightarrow$ mass)
12: b1.append($p \rightarrow$ velocity+$\frac{\Delta T}{2} \cdot a2[i]$)
13: b2.append((envForce + collForce+friction + cumulative springs force)/$p[i] \rightarrow$ mass)
14: c1.append($p \rightarrow$ velocity+$\frac{\Delta T}{2} \cdot b2[i]$)
15: d2.append((envForce + collForce+friction + cumulative springs force)/$p[i] \rightarrow$ mass)
16: d1.append($p \rightarrow$ velocity+$\frac{\Delta T}{2} \cdot c2[i]$)
17: d2.append((envForce + collForce+friction + cumulative springs force)/$p[i] \rightarrow$ mass)
18: $p[i] \rightarrow$set velocity($p[i] \rightarrow$ Velocity+$\Delta T \cdot b2[i]$)
19: $p[i] \rightarrow$ set position($p[i] \rightarrow$ position+$\Delta T \cdot b1[i]$);
20: **end for**

The lines 1 to 8, simply set up some vectors, their complexity is constant. Afterwards we have a for loop, from line 9 to 20, on all of the particles of the system. In this for loop every operation is a constant time operation except for the computation of the springs forces, lines 11, 13, 15 and 17. The "cumulative springs force" is a loop over every spring attached to that particle. For each particle the springs forces are computed four times.
The total complexity of this algorithm then is O(|particles| + |springs|).
The number of particles and springs are related, but this relation differs between the cloth and the other two objects. The sphere and cube connect every two particles with a spring, then the relation between the two is:

$$|springs| = \frac{|particles| \cdot (|particles| - 1)}{2}$$

The cloth on the other hand connects only close particles with a spring. In this case the relationship between the two is:

$$|springs| = (width - 2) \cdot 4 \cdot (height - 1) + 3 \cdot (height - 1) + 2 \cdot (height - 1) + (width - 1)$$

Where the width and height are the width and height of the cloth expressed in number of particles. Since in the benchmark we considered only square cloths, then:

$$width = height = \sqrt{|particles|}$$

It follows that for a large number of particles:

$$|springs| \approx 4 \cdot |particles|$$

From the results of the benchmark we can observe that updating the sphere and cube is expensive even for a small number of particles while the cloth allows for many more. This is explained by the relationship between particles and springs in the different objects.

8.2 Limitations

I have not implemented intra object collision, this is not a problem for the deformable cube and sphere, since they can not deform enough to collapse on themselves, but it is noticeable for the cloth. Since the cloth can bend on itself there are some situations in which it will self penetrate and it is fairly noticeable. Another limitation is given by the integrators. Even though the fourth grade Runge Kutta integrator is quite good it still requires a fairly small time step. If there are multiple deformable objects on screen at once (or a single deformable object formed by many particles), then to update all of them using the fourth grade Runge Kutta integrator can be quite expensive. It is then necessary to have a small number of objects at once on screen and they can not be formed by an excessive number of particles. There are only three simple solid objects. It could be interesting to implement more. Alternatively, when a 3D object is implemented, the user could be prompted to choose whether it should be a solid or soft object.

8.3 Future works

To further improve the simulation, it becomes critical to implement better integrators. For example, the Implicit Euler integrator or the Verlet integration scheme. As said above, self collision should be implemented, since it would make the cloth more realistic. I have implemented BVH to improve the performance, but I have used spherical BVH for every object. Different BVH could be implemented, such as object oriented bounding boxes, convex hulls and discrete oriented polytype. This would make the BVHs more flexible, allowing them to more tightly enclose the objects. Another popular technique in computer graphics to improve performance of collision detection is space partitioning, in which the space is subdivided in subspaces and collision detection is run only if two objects are part of the same subspace. Space partitioning could be added to the simulation to improve its performance.

References

[1] `https://simple.wikipedia.org/wiki/Friction`.

[2] opengl-tutorial. `http://www.opengl-tutorial.org/`.

[3] F. Ding. Rapid cloth collision detection using bounding sphere trees.

[4] B. H. M. Teschner1, S. Kimmerle2. Collision detection for deformable objects.

[5] D. J. N. T. Matthias Müller, Jos Stam. Real time physics, class notes.

[6] A. B. Nur Saadah Mohd Shapri, Riza Sulaiman. Collision detection between cloth and a solid object using mass spring model and bounding volume hierarchy.